下载APP

26 | 进程是如何使用操作系统内存的？

2022-02-23 于航

《深入C语言和程序运行原理》 课程介绍 >

讲述：于航

时长 15:59 大小 14.64M

你好，我是于航。

对于计算机软件的正常运作，内存（Main Memory）所发挥作用的重要性不言而喻。无论是处在“幕后”的操作系统，还是位于“台前”的用户应用程序，它们在运行时都会将所需数据从磁盘等外部存储器转移至内存。实际上，内存和 CPU 芯片上的 L1、L2 等高速缓存，一同构成了计算机中用于支撑程序高效运行的缓存系统。

领资料

今天，我们会先从整体的视角看看内存在计算机系统中的作用，然后再一起探究进程是如何在操作系统的控制下与计算机内存交互的。

计算机内部的缓存系统

通常，文件会被存放在容量较大的磁盘中。但磁盘作为一种提供数据持久化存储的设备，采用了机械式的数据寻址方式，这就使得它**无法匹配 CPU 在完成相关操作时，所需数据在访问速度上的要求**。而内存则以快于磁盘几万甚至十几万倍的读写效率，承担起了与 CPU 直接交互的重任。

但随着摩尔定律的不断应验，CPU 与内存两者在数据访问效率的“供需关系”上又出现了问题。因此，现代计算机通过在这两者之间引入更多读写速度更快，容量却更小的高速缓存层，并基于局部性原理，让 CPU 经常使用到的数据可以被更快地再次访问。通过这种方式，**由 L1、L2 等片上高速缓存，以及内存组成的缓存系统，便成为了计算机中用于承载应用运行时数据的主要部件**。

网上有一个十分形象的例子，描述了内存在整个计算机系统中的“地位”：假设你作为负责人，每天都在一间专属办公室里处理各种事务。办公室内有一个硕大的档案柜，里面存放着你有权限接触的所有办公材料。当每一次需要处理某个具体事务时，你都会首先将需要的相关材料一次性地从档案柜中全部拣选出来，并将它们陈列在书桌上，然后再继续进行处理。而随着时间的推移，那些经常被翻阅的、相关性较强的材料会被摆放在距离你手边较近的位置，而相关性较弱的材料则会被放在较远的位置。

将这个例子类比到计算机，你会发现两者之间有着类似的行为模式。每一个程序的机器代码（事务）在可以被 CPU（负责人）正常执行前，操作系统都需要先将它们从磁盘（档案柜）“搬移”到内存（书桌）中。而随着程序的不断运行，那些被经常访问的数据便会被存放到较高级别的缓存（较近位置）中。相反，不常用的数据则会被存放在较低级别的缓存（较远位置），甚至驻留在内存中。

但是，对计算机而言，程序与内存之间的交互细节远比“办公室日常”要复杂得多。现代操作系统会同时执行十几个甚至几十个程序。因此，如何从有限的内存中合理地为它们分配所需资源，并同时兼顾安全性、高效性，便成为需要考虑的首要问题。

现代计算机通过名为“虚拟内存”的机制，做到了这一点。下面，我们来进一步看看这个机制的具体工作原理。

虚拟内存机制

顾名思义，虚拟内存（Virtual Memroy）对应于物理内存（Physical Memory）。其中，前者是由操作系统抽象出来的一个概念，它在后者的基础之上进行了一层抽象，以帮助运

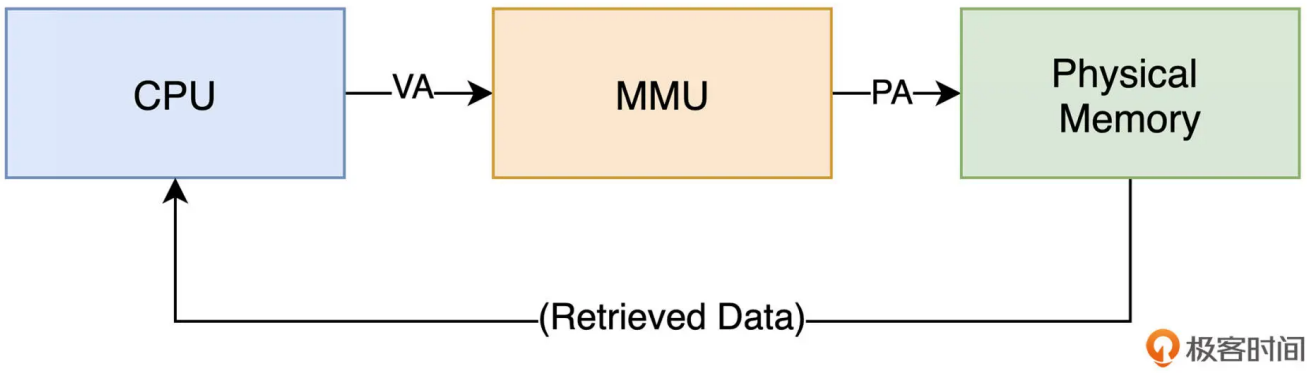
行于其上的应用程序合理地分配内存，并管理内存使用。

因此，如下面的代码所示，我们在应用程序中打印出的各种指针值，它们实际上都对应于虚拟内存中的某个地址，而非实际的物理内存地址（Physical Address，PA）。这些地址被称为“虚拟地址（Virtual Address，VA）”。所有程序可以使用的虚拟地址则构成了虚拟地址空间（VAS）。

复制代码

```
#include <stdio.h>
int main(void) {
  int x = 10;
  printf("%p", &x);  // 0x7fff32cf54fc.
  getchar();
  return 0;
}
```

CPU 在访问内存中的数据时，会借助其芯片上名为“内存管理单元（MMU）”的硬件，首先将虚拟地址动态翻译为对应的物理地址，然后再进行实际的数据获取。你可以通过下图来直观地理解这个过程。



虚拟内存机制的一个最重要特征，就是**为每一个应用程序进程都抽象出了独立于物理内存的虚拟地址空间**。这意味着，从进程的角度来看，它可以独享整个计算机上的所有内存。现代操作系统通常采用 32 或 64 位地址空间，两者分别拥有 2^{32} 与 2^{64} 个地址。通过这种方式，编译器在构建应用时，便不需要考虑各二进制数据段应该被实际加载到内存中的何处，所有应用均可使用统一的静态文件结构。

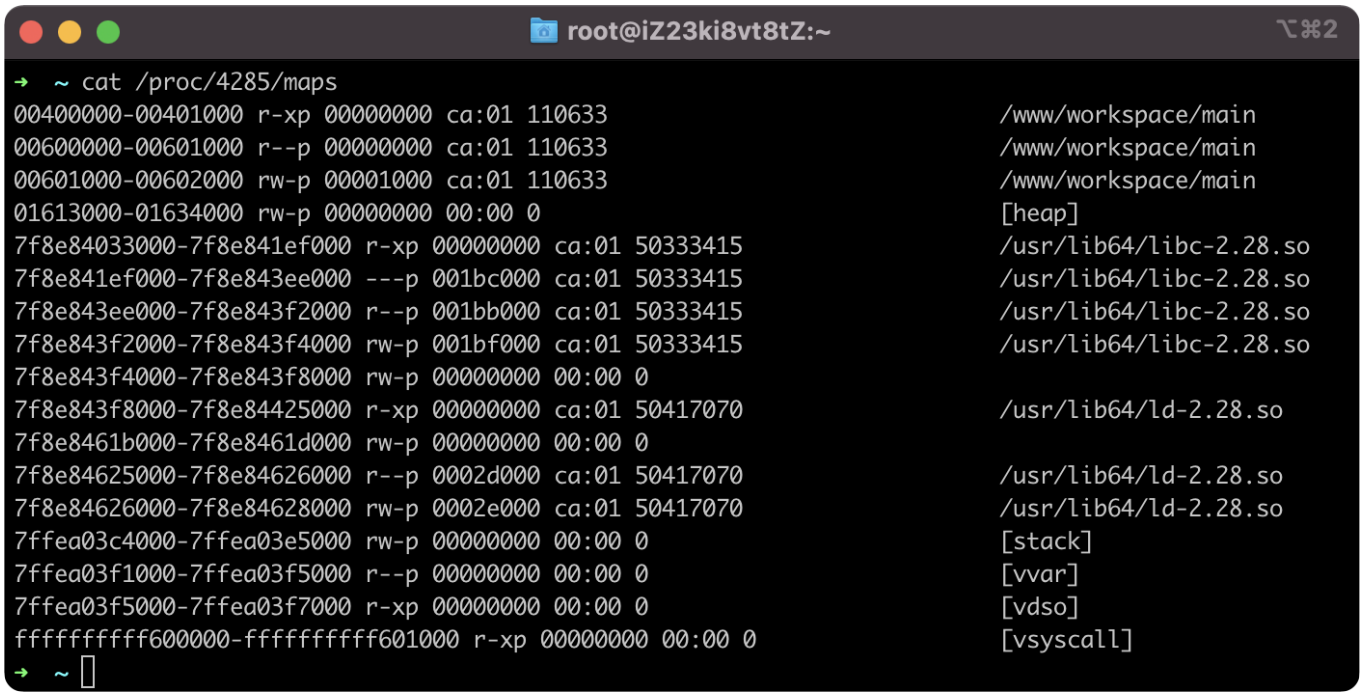
比如，在 64 位 Linux 系统中，与应用代码相关的 Segment 会从 VAS 的固定地址 0x400000 处开始加载。而其他 Section 内容将在满足一定对齐要求的情况下，按顺序被

连续加载到高地址方向的虚拟内存中。这样，无论是程序在二进制文件内的静态视图，还是被加载到 VAS 后的运行时视图，它们都可以在虚拟内存的隔离下，在表现层有着稳定一致的布局。

而通过下面这行命令，我们便可以查看某个运行进程的 VAS 布局情况。注意，其中的 “” 需要被替换为进程对应的 ID。

```
1 cat /proc/<pid>/maps
```

这里，我们尝试将本小节开头处的那个 C 程序的 ID 替换到上述命令中。在运行该命令后，可以得到如下图所示的输出结果。

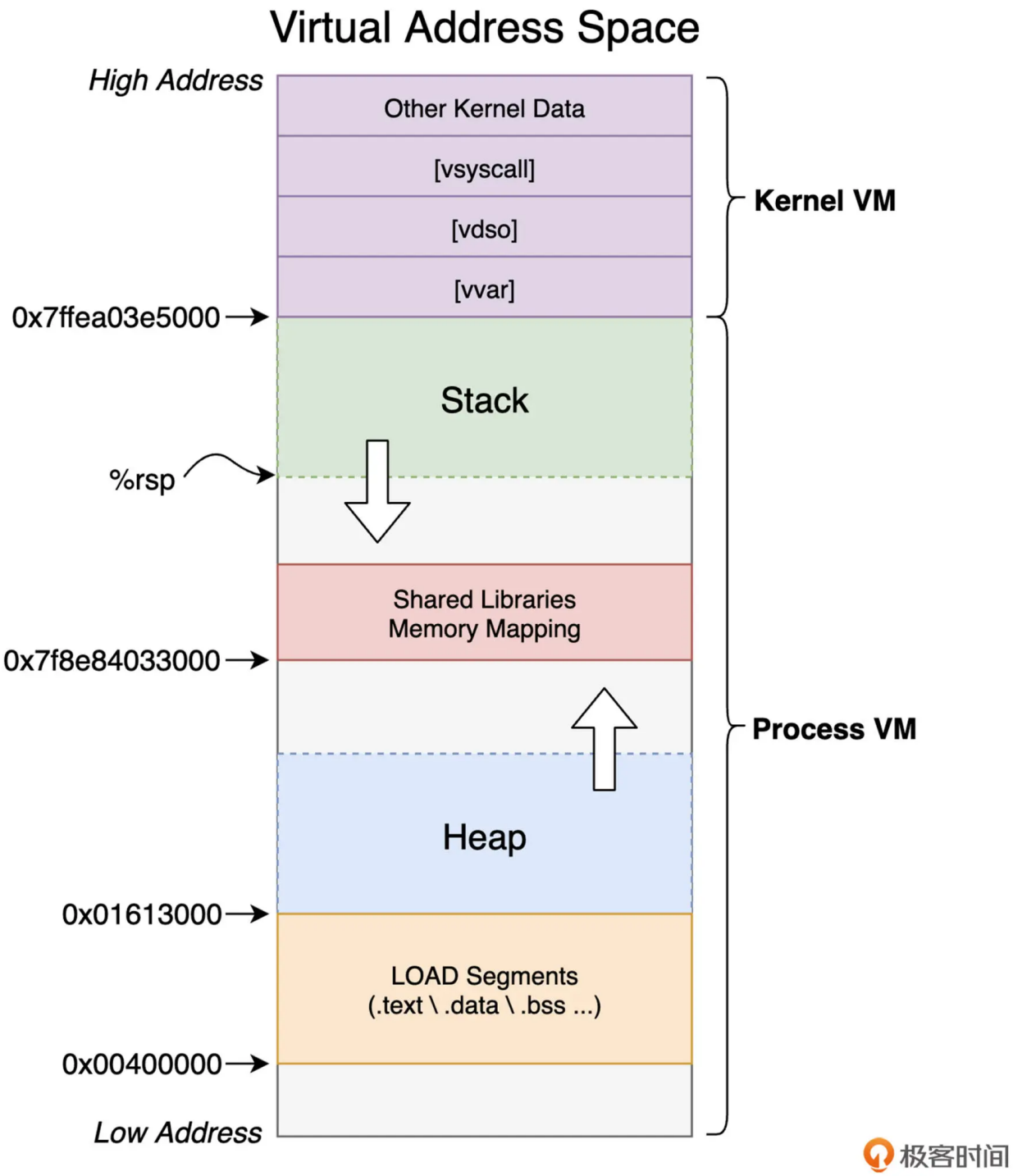


这里，命令按照地址由低到高的顺序打印出了进程 VAS 内，每一块已经被占用的连续虚拟内存地址，对应的映射信息。如最右侧一列所示，这些内存中的内容或是来自于某个具体文件（/www/workspace/main），或被用作其他用途（[heap]）。

VAS 中的数据布局

将上图中的信息进行归类，我们可以得到如下图所示的 Linux 进程在 VAS 内的统一数据布局结构。这里我根据类别，将不同的数据用不同的颜色进行了标注。并且，为了方便你找

到这两个图之间的内存段对应关系，我将上图中的一些关键性地址信息也选择性地标注在了下图中。



总的来看，Linux 进程 VAS 中的数据，按照地址由低到高的顺序，可以被分为下面这几个主要部分。

LOAD Segments：这部分数据被加载到从地址 0x400000 开始的虚拟内存中。其主要内容为应用程序 ELF 二进制文件内定义的各种 LOAD Segment 结构。按照顺序，与代

码相关的 Text Segment（包含 .text、.rodata 等多个 Section）位于最低地址处，紧接着为包含有已初始化和未初始化数据的 Data Segment（包含 .data，.bss 等多个 Section）。

堆（Heap）：关于堆内存，我已经在 08 讲 中介绍过。随着动态数据的不断产生，它将向 VAS 的高地址方向不断增长。

共享库数据：这部分内存中包含有与各类 .so 共享库相关的数据，程序会在运行时通过动态链接器来完成对它们的加载和处理。我会在第 29 讲再为你详细介绍。

栈（Stack）：关于栈内存，我已经在 05 讲 中介绍过。随着各种局部变量的不断产生，它将向 VAS 的低地址方向不断增长。

用于系统调用加速的内核数据：接下来的三个虚拟内存区域 [vvar]、[vdso]，以及 [vsyscall] 中包含有操作系统内核的代码和数据结构，它们主要提供了用户进程可以直接与内核进行交互的接口。其中，[vvar] 中包含有只读的内核数据。而另外的 [vdso] 与 [vsyscall] 则包含有用于辅助操作系统，加速用户进程执行某些系统调用过程的信息。

其他内核数据：除此之外，在进程 VAS 的高地址处，还可能包含有与当前进程相关的各种数据结构。甚至，该区域内的某段虚拟内存页还会被直接映射到某段被所有进程共享的物理内存页上（比如用于 MMIO）。

可以看到，得益于虚拟内存机制的抽象，进程可以使用完全统一、独立的内存数据布局，而不用考虑这些数据在真实物理内存中的具体存储细节。那么，虚拟内存机制究竟是如何对物理内存进行管理的呢？接下来我们具体看看。

使用页表维护虚拟页状态

为了保证效率，操作系统通常会以“页”为单位，来在磁盘与内存之间传递数据。而实际上，它也正是通过为每一个进程提供独立的“页表”结构，来维护 VAS 中的虚拟页在对应物理内存中的映射状态的。

页表本身被维护在物理内存中，其内部由众多的“页表项（Page Table Entry，PTE）”组成。进程 VAS 中的每个虚拟页都对应于页表中的某个 PTE，而 PTE 中则包含有用于描述该虚拟页状态的众多字段。每一次 MMU 需要将一个虚拟地址翻译为物理地址时，它都会首先读取页表，以查询相关的 PTE 信息。然后，再根据虚拟地址内隐含的偏移信息，找到对应页中的目标位置。

在简化的实现中，PTE 可能由一个“有效位”字段与一个“地址”字段组成。其中，有效位用于表明该虚拟页是否已被缓存在物理内存中。若该位置位，则地址字段中存放有该页在物理内存中的起始位置。而在该位复位的情况下，若地址字段为空，则表明该虚拟页还未被分配。否则，地址字段中便保存有虚拟页内容在磁盘上的起始位置。

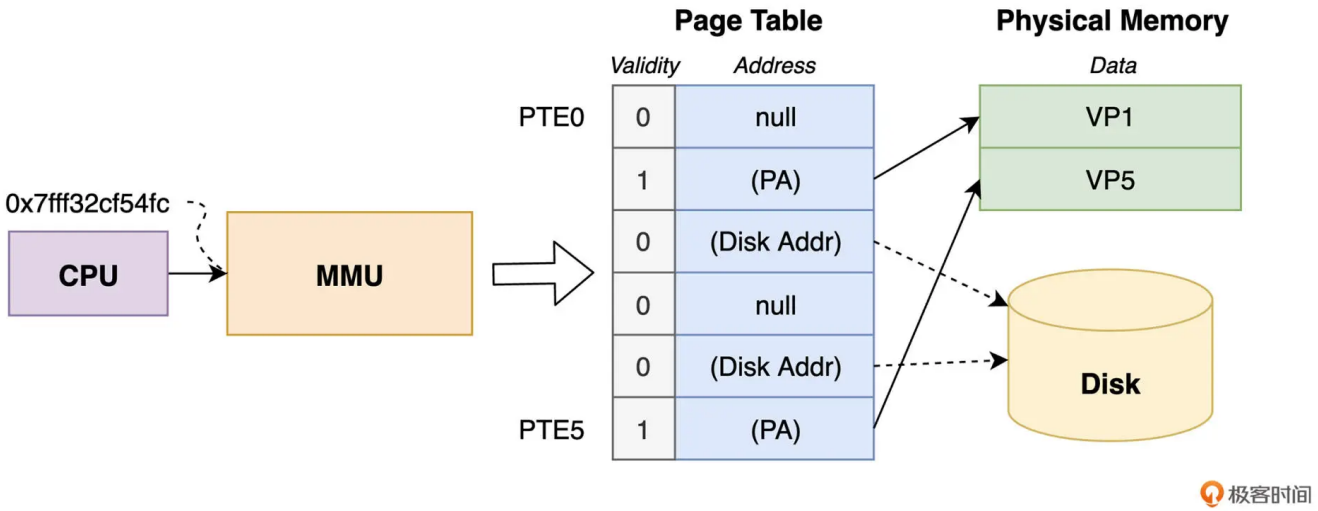
当 CPU 需要访问某个虚拟地址上的数据时，通常会发生以下两种情况：

MMU 查找进程页表，发现目标数据已被缓存，进而直接通过 PTE 中的物理地址获取并返回所需要的数据；

MMU 查找进程页表，发现目标数据未被缓存，此时它会触发一个“缺页异常”。

在第二种情况中，缺页异常将会调用内核中特定的异常处理程序，该程序会在物理内存中选择一个页，以用来承载当前虚拟地址所对应的物理数据。其中，对于空闲页，内核会直接将虚拟页对应的内容从磁盘拷贝到该物理页中；而对于非空闲页，若该页已经被修改，则内核会首先将它的内容换出，即更新到磁盘。然后，再将磁盘上的内容拷贝至这块物理页中。

这里你可以先暂缓脚步，通过下图来直观地理解 CPU、MMU、页表、物理内存，以及磁盘五者之间的协作关系。



页表隔离了进程的 VAS 与物理内存，使得两者之间的映射关系变得更加自由。而在这种方式下，当不同进程使用不同页表维护其各自 VAS 中虚拟页的映射时，多个进程之间便可做到真正的数据共享。而我将在 29 讲中介绍的“动态链接”技术便以此为基础。不仅如此，独立的 VAS 与页表也使得进程之间的私有内存不会被相互访问。

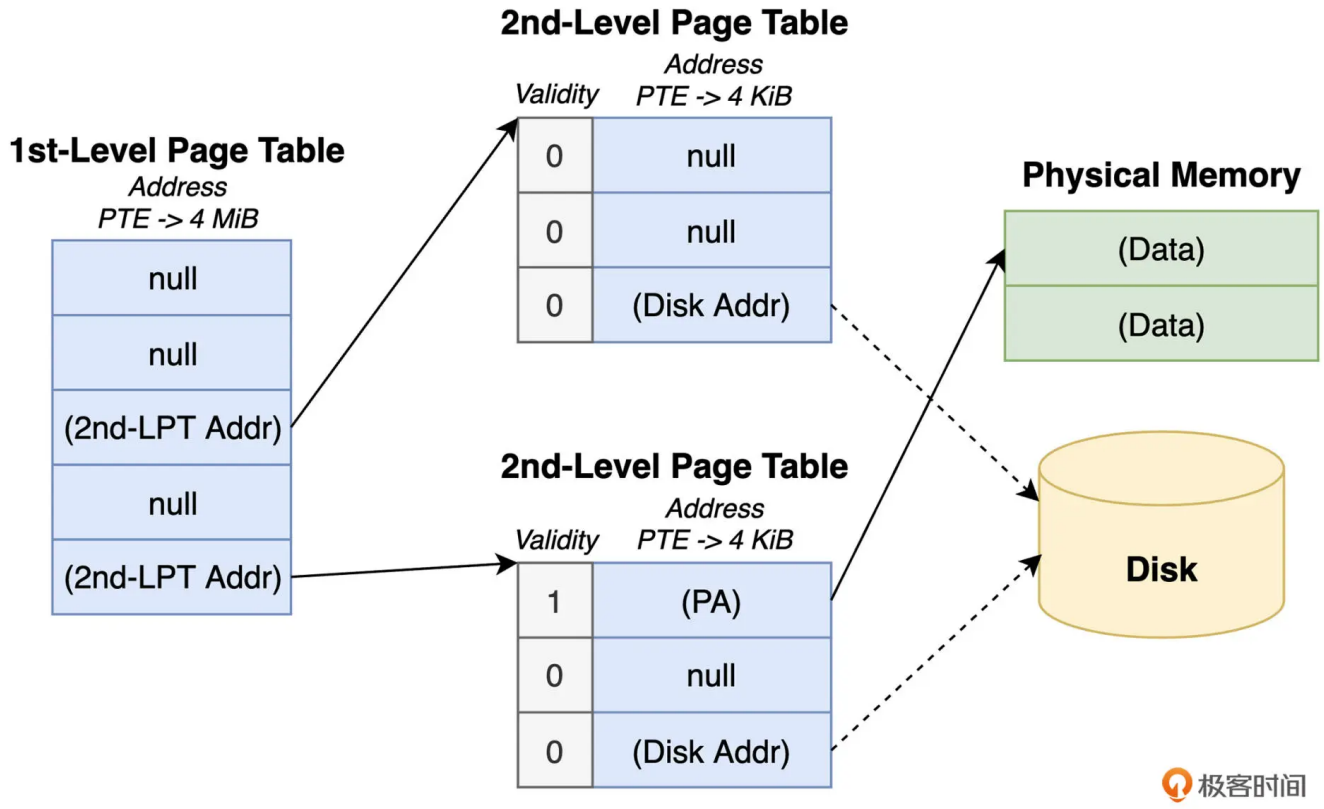
另外，通过在 PTE 中增加用于访问控制的相关字段（如可读、可写、可执行），CPU 可以在程序尝试非法访问某块内存数据时做出异常响应。

使用多级页表压缩页表体积

但是，上面介绍的一级页表有时却可能无法满足需求。试想，以目前常用的 64 位地址空间为例，假设页大小为已知最大的 2MiB，为保证完整映射，每个 PTE 大小为 8 字节。而为了能够在单一页表内维护进程整个 64 位 VAS 中所有虚拟页的信息，那么便需要为其匹配一个大小为 65535 GiB 的页表，而这显然是不现实的。因此，现代计算机通常会采用“多级页表”的方式，来优化页表的大小。

多级页表的思路很简单。以二级页表为例，假设在一个 32 位地址空间中，页大小为 4KiB，每个 PTE 大小为 4 字节。此时，MMU 在进行物理地址查询时，首先会根据虚拟地址中隐含的虚拟页号信息来查找一级页表内的目标 PTE，而一级页表中的每个 PTE，此时实际上负责映射 VAS 中的一个 4MiB 的片。

按照树的形式展开，每个一级页表也都对应着一个独立的二级页表，二级页表中的每一个 PTE 则负责映射当前一级页表的 4MiB 的片中的某个 4KiB 的块。当一级页表查询完毕后，MMU 便可得到指向目标二级页表的地址。通过该地址，再联合虚拟地址中的另一部分虚拟页号信息，它便可找到目标数据所在物理内存的具体页。最后，结合虚拟地址中的页偏移信息，目标数据的最终物理地址便可被成功地“转换”出来。你可以通过下图来进一步理解上述流程：



多级页表可以节省内存空间的两个最重要因素是：

当一级页表中的某个 PTE 没有实际映射时，其对应的二级页表便不会被创建；

只有一级页表才需要常驻内存，二级页表可以仅在需要时再创建，或从磁盘调入。

实际上，上述二级页表的使用形式可以被推广到任意的 N 级。但总体来看，页表的级数并非越多越好，因为更多的页表级数也就意味着更长的物理地址查询时间。目前常见的多级页表为 4 级，而在 Ice Lake 等处理器中，也出现过 5 级页表。

使用 TLB 加速 PTE 查询

多级页表虽然可以压缩页表占用的内存量，但用 MMU 进行页表的逐级查询，这个过程也并不是毫无成本的。现实情况中，计算机通常会结合使用名为“翻译后备缓冲器（Translation Lookaside Buffer，TLB）”的硬件设备来加速这一流程。

TLB 属于 MMU 的一部分，它可以加快 MMU 根据虚拟地址查询 PTE 的过程。你可以将 TLB 理解为一个简单的具有 N 行 M 列的矩阵，MMU 会从对应虚拟地址中提取出用于查询表项的 TLB 索引与 TLB 标记。这两个值可以联合起来使用，并定位到 TLB 中的一个具体单元格。而此时，若该单元格内有值，则 MMU 可以直接使用该值，来与虚拟地址中的

其他信息一起组成最终的物理地址。否则，MMU 仍然需要通过逐级查询页表的方式来获取目标页的物理地址。

再谈共享对象与私有对象

上面，我曾在“使用页表维护虚拟页状态”一节的最后提到，借助虚拟内存机制，不同进程之间可以共享物理内存上的同一段数据。

这些数据在物理内存中实际存放时，可能并不是连续的。而借助于页表实现的“虚拟页与物理页映射关系分离”，我们可以确保 CPU 能够按照连续的方式来使用这些数据。

而当某个共享进程试图对这些共享数据进行修改时，操作系统便会通过“写时复制（Copy-on-write）”的方式，来将被变更数据所在的物理页进行复制，并通过修改页表，来让修改进程可以私有化这部分数据。

总结

这一讲，我主要为你介绍了进程是如何在操作系统的控制下使用内存资源的。

在现代计算机中，内存和 CPU 芯片上的高速缓存一起构成了用于承载应用运行时数据的缓存系统。而这个缓存系统，在名为“虚拟内存”机制的帮助下，能够以一种更加优雅的方式运作。

虚拟内存机制为每一个进程都抽象出了独立且私有的虚拟地址空间（VAS）。VAS 中使用虚拟地址进行寻址，当 CPU 需要通过该地址访问内存中的某个数据时，芯片上的内存管理单元（MMU）会将该地址转换为对应的物理地址。不同的操作系统都会在 VAS 中为进程使用相对统一的数据布局方式，这样，编译器便可简化其构建应用的流程。

操作系统使用名为“页表”的数据结构来维护 VAS 中虚拟页与物理页之间的映射关系。通过查询页表项（PTE）的状态，操作系统可以直接获得目标数据所在页的物理地址，或是通过触发缺页异常，来让操作系统内核将目标数据从磁盘加载到物理内存中，然后再重新获取该地址。在这个过程中，内核可能会将物理内存中，某个已修改的非空闲页的内容换出到磁盘。

为了减小分配给每个进程的页表大小，现代计算机通常采用多级页表的方式来管理虚拟页与物理页的映射关系。而在这种方式下，由于需要查询的表项过多，计算机还会采用名为 TLB 的硬件设备，来缓存之前的表项查询结果，并加速下一次相同虚拟页的查询过程。

最后，虚拟内存机制使得多个进程可以同时共享物理内存中的某段数据，而无需将数据拷贝多份。但当某个进程试图修改这些共享数据时，操作系统会通过“写时复制”的方式来将被修改数据进行拷贝，并使其对修改进程私有化。

思考题

试着查阅资料来了解一下，为什么 Linux 进程 VAS 从地址 0x0 开始，直到 0x400000 的低地址段，没有存放任何数据？欢迎在评论区告诉我你的发现。

今天的课程到这里就结束了，希望可以帮助到你，也希望你在下方的留言区和我一起讨论。同时，欢迎你把这节课分享给你的朋友或同事，我们一起交流。

分享给需要的人，Ta订阅超级会员，你将得 50 元

Ta单独购买本课程，你将得 20 元

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 25 | 可执行二进制文件里有什么？

下一篇 27 | 编译器在链接程序时发生了什么？

更多学习推荐

2021 最新大厂
Go 工程师面试真题

大厂面试真题 + 金九银十全新整理 + 核心知识全覆盖

限量免费领取

精选留言 (2)

写留言

LDxy
2022-02-23

位于虚拟地址空间的最低部分，未赋予物理地址。任何对它的引用都是非法的，用于捕捉使用空指针和小整型值指针引用内存的异常情况。

1

Simon
2022-02-24

64位系统的地址空间只有 2^48，高十六位都是0，共256T。

> 作者回复: 没错，对于大多数体系比如 x86-64 和 ARMv8 来说是这样的，但也有支持完整 64 位虚拟地址的体系，比如 Oracle SPARC 的某些型号。